HEWLETT PACKARD

HP 83000
Digital Test
System

Command Interface

HP 83000 Digital IC Test System

Command Interface

HP SmarTest

SERIAL NUMBERS

This manual affects all systems with software revisions up to 2.0

HEWLETT PACKARD

HP Part No. 83824-91014
Printed in the Federal Republic of Germany April 1990

Revision 2.01

Legal Information

Notice

The information in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, perfo on, or use of this manual.

Warranty

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Manuals available in this Documentation Set.

About Using the HP 83000 model F250 Production Test Software:

1. Developing a Production Test Program	E2824-90011
2. Preparing and Executing a Production Test	E2824-90012
3. Understanding the Applications Model	E2824-90013
4. Command Interface Reference	E2824-90014

About Using the HP 83000 model F330:

1. Interactive Use of the HP 83000	E2823-91011
2. Standard Test Function Reference	E2823-90012
3. Firmware Command Reference	E2823-91013
4. Programming Reference	E2823-91014
5. Using the CAS Link	E2823-90011

About Installing and Servicing the HP 83000 model F330:

1. HP 83000 Site Planning and Preparation Guide	E2823-91001
2. Installing the HP 83000	E2823-91002
3. Maintaining the HP 83000	E2823-91003
4. Servicing the HP 83000	E2823-91004

Preface

Purpose

This manual gives reference information about the Command Interface for the production software for the HP 83000 Digital Test System. The purpose of the manual is to provide enough information to allow you to use the Command Interface to meet your application. The prerequisites for using this manual are:

- an understanding of HP-UX/UNIX
- a working knowledge of C language
- familiarity with the UNIX mb tool.

Contents

Figures

1

Introduction

The Command Set

A Command Set is a set of user-written commands that you can call from the application model file. The calls are directed to the Command Set named under the icon in the Test Program page of the Data Manager. Calls to Command Sets are as follows:

- D_BRIDGE(—), to execute a command located in the Command Set named under the DataBridge icon
- EXEC_INP(—), to execute a command located in the Command Set named under the ExecutableInput icon
- PBS_BRGE—), to execute a command located in the Command Set name under the ProbeSimulator icon.

It is also possible to call a Command Set named under the uexproc icon in the Data Manager. Calls to this Command Set are made directly from the testflow, but otherwise are treated the same as other Command Sets.

If there is no name under the icon, indicating that you have not selected a Command Set, then execution of the command automatically fails. This failure generates an error message and manual intervention is required to rectify the situation. For such an error, a RETRY button and an ABORT button are made available.

Command Call Syntax

The syntax of a command call written in the application model file is as follows:

```
<c  > '=' <set_identifier> '(' <command_n,p  > ')'

<set_identifier> ::= D_BRIDGE_CALL | PSB_END_C   |   _   .CALL

<command_and_param> ::= <   >  [<opc><   ]...[R]

<rar>  ::= [a-zA-Z1-9!*$]*++.,/:<>[\]^_{|}*...
<cmd>  ::= [a-zA-Z1-9!*$]*++.,/:<>[\]^_{|}*...
<opc>  ::= [  non | + ],...
<param>  ::= [a-zA-Z1-9!*$]*++.,/:<>[\]^_{|}*...
```

In the following example, the co d prompt/text input is executed. The command implementation is b d in the Command Set as selected in the Data Manager under the EXEC_INP item. The command is given the argument NAME. Any result generated by execution of the command is assigned to the ModelFile variable inputname (see "Writing a Function" in Chapter 2 for details).

```
' inputname = EXEC_INP(prompt/text NAME);
```

Directory Paths

There are four items in the Data Manager from which you can select Command Sets. The Command Sets reside in the the following default directories:

UserProcedure : /hp83000/userproc_libs

ProberHandler : /hp83000/79_libs

DataBridge : /hp83000/data_bridge

ExecutionInput: /tp83000/execution_input

You can use your own directory paths and names instead of the defaults.

To install a new Command Set, first create a new directory under the directory path you want to use. When you do this, the new Command Set will be accessible from the Data Manager. You will need to put the directory path in the Device Path box of the user_proc, exec_input, prober handler and data bridge windows of the Data Manager if you are not using the default directory.

For example, to install a new command set called

```
newcp
```

in the user procedure library in /hp83000 , type the following:

```
cd /hp83000/userproc_libs
mkdir newcp
cd newcp
```

When you have created the new directory, and then used the cd command to move to it, you can proceed with installing the Command Set as defined in Chapter 2, Chapter 3, and Chapter 4.

Command Set Types

A Command Set can be one of the following types:

- a set of user written functions (see Chapter 2)
- a set of programs or scripts (e Chapter 3)
- a VBE instance (e Chapter 4).

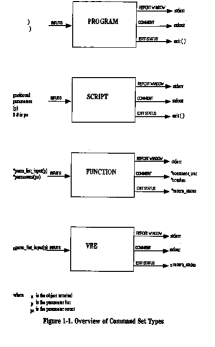
For a program or a t, you can either wait for it to terminate before executing the next and, or start it without waiting for it to terminate (equivalent to the IX 4).

Chapter 2, Chapter 3, and Chapter 4 explain how to implement the diff t Command Set types, assuming that you are located in the newly d directory where the Command Set is to be installed.

Every command that you implement, irrespective of the Command Set type, receives parameter inputs, and parameter counts. Similarly, from each implemented command it is possible to:

- send messages to the Report Window
- assign a string (comment) / datalog a interprocedure string
- and to define the success or failure of the command execution (exit status).

How this is done is different for each type of Command Set and is explained in the relevant chapters. Figure 1-1 summarises how this is achieved for each d Set type.



where
- a is the object invoked
- b is the parameter list
- c is the parameter record

Figure 1-1. Overview of Constrained Set Types

2

Function Command Set

A Function Command Set is a set of user written functions. This chapter describes how to:

- install a Function Command Set
- write a Function.

A list of Function Command Set error me is given in "Function Command Set Error Messages" in Appendix A.

Installing a Function Command Set

To install a function Command Set:

1. Type the following at the command line:

   /bp5000/prod_sw/bin/ tsll_cs -t

   The tsnll_cs script copies the following into your

   a special Makefile

   the function template fn_templ.c

   and a command interface link.

2. Use the template file fn_templ.c (see " ction Template File" in Appendix B) for each source file in which you want to implement the functions of the Command Set.

   You can write all of your functions into one source file, or split them up into several source files. In both cases, use the template provided. You can ignore the template and generate your own source files. However, this is not recommended, because the template file uses the exact syntax of the mandatory function declaration.

3. Code your functio (as described in "Writing a Function").

4. Type

   make depend all

   at the command line. This action generates the shared library *libcfnt.sl*. This is the file that is actually loaded by the command interface to make your functions available to the production software.

You can call the tsnall_cs script with the -u option. This means that a user procedure function template is copied into the directory instead of the normal fn_templ.c.

The tsnall_cs -t FUNCTIN can be called as often as required and does not destroy any source files you have created. It does, however, always overwrite the Makefile and the template file fn_templ.c, so do not implement any of your functions in a file with these names.

The make depend command updates the Makefile to make the newly created source files visible to the Makefile. You only need to call make depend when a new source file has been created.

Use the make all command as often as necessary until you get a clean compile.

The implementation is complete when the shared library *libcifjat.sl* exists within the directory.

Function coding must adhere to the rules as defined in the section "Dependencies".

Writing a Function

The function must be declared as follows:

```
void FunctionName (
                   char *parm_list_input,
                   int  parmcc  t,
                   c    *comment_ t,
                   int  *comlen
                   int  * turn_status
                   )
```

where *FunctionName* is a unique function identifier in the Command Set.

The parameters for the declaration are as follows:

*parm_list_input	is a pointer to a character array of length CI_MAX_PARM_INPUT + 1. This array holds all parameters directed at the function, and contains a list of 0 or more tokens which are separated by a space or a tab.
*parmcount	indicates the number of parameters present within the p _list_input array. The length of each ter may vary, and could even be 1 parameter occupying the whole parm_list_input array.
*comment_out	is a pointer to a character array of length CI_MAX_COMMENT_LEN + 1. y comment that is to be assigned to the application model variable must be written to this array and is p on automatically, depending on the value of the comlen parameter; this does not , however, to user procedures.
	is a pointer to an int, and must be set to the length of the comment written to array *comment_out.
*return_status	is a pointer to an int, and must be set to indicate the success or failure of function execution. See Chapter 5 for an explanation of the valid codes available.

Direct a Message to the Report Window

To direct a message to the report window from a function, use the stderr.

Dependencies

When you code a Command Set function, consider the following dependencies:

- A maximum CI_MAX_COMMENT_LEN bytes are written to command_out.

  Failure to observe this dependency corrupts the command interface. This may result in a segmentation violation or a bus error.

- A maximum CI_MAX_COMMENT_LEN bytes are read to comment.

  Failure to observe this dependency results in a comment truncation warning if the value is set greater than CI_MAX_COMMENT_LEN, or results in 0 bytes being transferred to the application model file variable if the value is negative. If the comment length is greater than the comment itself, but is still within the limits of CI_MAX_COMMENT_LEN, random garbage is assigned to the application model file variable.

- The return_stat is set one of the following constants:

  CI_CALL_PASS = 0

  CI_TEST_PASS = 0

  CI_TEST_FAIL = 1

  CI_CALL_ERROR = 2

  CI_CALL_BREAKD = 3

  Failure to observe this dependency results in an undefined return status being returned to the production software. This results in zero bytes of comment being transferred to the application model file variable. The undefined state is treated as an error and an error message is generated. Manual intervention is required and a [RETRY] and an [ABORT] button are made available.

- Any timeout requirements must be satisfied by the user (SIGALRM available), as HP SmartTest simply waits for the function to finish.

  Failure to observe this dependency may result in a production software hangup.

- The Makefile ded by the software must be used for source file compilation.

  Failure to observe this dependency may result in a run time crash. This is caused by unresolved dynamically created external references from the shared library referencing unlinked Command Set Programming Interface (CPI) functions. The Makefile also contains special commands to switch between debugging/non debug modes of operation.

- Error M s are your responsibility, and must be written to stderr. This is the medium for passing messages to the report window.

- The actions for all SIGNALS remain unchanged for the duration of the function execution. The exceptions are SIGALRM, SIGINT, SIGUSR1 and SIGUSR2.

  Failure to observe this dependency causes system hangup and/or crash.

Additional Information

CPI Functions

CPI functions are obtained if you include the file libcipi.h at the beginning of each source file. This is no ssarily included in the fn_temple.c template file. Further information about CPI functions is given in Chapter 6.

Binding a Different Library to the Command Set

If you use a function defined in a library other than libcipi.a/.so, you must add a line in the Makefile to specify the library.

To do this:

1. Open the Makefile
2. Add the full path of the library to the LIBS line in the Makefile
3. Save the Makefile
4. Type make clobber t at the command line.

For example, to use a function defined in the library lib_x.a, located at /hp63000/user_libs/, the Makefile must contain the line:

```
LIBS = /lb 30  /user_libs/lib_x.a
```

3

Program Command Set

A Program Command Set is a set of programs or scripts. This chapter describes how to:

- Install a Program Command Set
- write a program
- write a script

A list of error messages for a Program Command Set is given in "Program Command Set Error Messages" in Appendix A.

Installing a Program Command Set

To install a program Command Set:

1. Type the following at the co d line:

   /bp8000/prod_aux/lbin/install_cs -t PROGRAM

The install_cs script copies the following into your directory:

   a special Makefile

   the program template pg_temp1.c and the script template sc_temp1.c

   and a command interface link.

2. Copy the template file pg_temp1.c (see "Pro Template File" in Appendix B) to the set of source files in which you want to implement the program for the Command Set. Use the script template sc_temp1 (see "Script Template File" in Appendix B) to generate scripts. Each program has a unique source file. You can ignore the template and generate your own source files. However, this is not rec ended, because the template file uses the exact syntax of the mandatory pro declaration.

3. Code your programs (as described in "Writing a Program"), or code your scripts (as described in "Writing a Script").

4. Type

   make depend all

   at the command line. This action compiles each of the source files into a separate executable unit. These units are available to Command Interface when you run the production software. Ensure that all the scripts are executable by using the chmod +x scriptname command.

The install_cs -t PROGRAM can be called as often as required and does not destroy any source files you have created. It does, however, always overwrite the Makefile, the template file pg_temp1.c, and the template file sc_temp1.c, so do not implement any of your functions in a file with these es.

The make depend command updates the Makefile to make the newly created source files visible to the Makefile. You only need to call make depend when a new sourcefile has been created.

Use the make all command as often as necessary until you get a clean compile.

The implementation is complete when make succeeds without compilation errors.

Coding of programs or scripts must adhere to the rules as defined in the section "Program Dependencies", or "Script Dependencies".

Writing a Program

The program declaration of in must be defined as follows:

```
main(argc, argv)
char *argv[];
int argc;
```

The parameters are p to the pr through the argv array in the conventional UNIX/C' manner. The parameter count is available through int argc. Parameters are passed to the program exactly as written in the Command Set call and are available in the order specified (left to right).

The following line shows an example declarati :

```
EESC_INP(init one two three);
```

The program executed is ' ' is, and the p three as follows:

```
argv[0] = init
argv[1] = one
argv[2] = two
argv[3] = three
```

To start a program in background e p nter of the Command Set call, as shown in the fo ple.

```
EESC_INP(init one two three &);
```

Exit Status

The exit status of the program conveys the success or failure of the program execution, and must be set. For a full explanation of the available exit codes, see Chapter 6. The fol line shown an example:

```
exit(CI_CALL_PASS)
```

Passing a Comment

To pass a comment to the appli ' n model file v ' ', write to stderr. All output that you write to stderr is classified as the comment, and is terminated by the termination of the program. The length of the comment is set automatically, and is equal to the number of bytes written to stderr.

When a program is started in background mode, comments are ignored.

Direct a Message to the Report Window

To direct a message to the report window from a program, write to stderr.

Program Dependencies

When you create a program Command Set, consider the following dependencies:

- Max CI_MAX_COMMENT_LEN bytes are written to stderr.

  Failure to observe this dependency results in a comment truncation warning and possibly causes a program failure with the message ERROR: CI: child stderr Write on Pipe. This means the interface has read the maximum n r of bytes allowed and has closed the input medium.

- The program exit( ) has is assigned one of the following constants:

  CI_CALL_PASS = 0

  CI_TEST_PASS = 0

  CI_TEST_FAIL = 1

  CI_CALL_ERROR = 2

  CI_CALL_BREAKD = 3

  Failure to observe this dependency results in an undefined return status being returned to the production software. This means that zero bytes of comment are transferred to the application model file variable. The undefined state is treated as an error and an error message is generated. Manual intervention is required and a [RETRY] and an [ABORT] button are made available.

- Any timeout requirements must be satisfied by the user (SIGALRM available).

  Failure to observe this dependency codes an endless loop into the function and results in a production software hangup, except when the NOWAIT

option is used. In this case you are r sible for hanging background processes.

- The Makefile provided by the software is used for plation of the program.

  Failure to observe this dependency results in compilation complications due to undefined external references on the CPI functions. This cannot happen if the Makefile provided by the software is used for the compilation. The Makefile program produces an executable program from each .c file in the directory. The Makefile also contains special commands to switch between debugging-debug modes of operation.

- Unless you modify the Makefile (see "Writing a Program with more than one Sourcefile", only one source file is allowed per program.

  Failure to observe this dependency results in compilation complications, because the Makefile has been ed for the single source per program type of Command Set.

- Error Messages are your responsibility, and must be written to stderr. This i the medium for g messages to the report window.

- The actions for all SIGNALS remain undisturbed for the duration of the function execution. The exceptions are SIGALRM, SIGINT, SIGUSR1 and SIGUSR2.

  Failure to observe this dependency critically jeopardizes system robustness.

- If a program forks another process then that process inherits the comment and report passing communication mechanics. All sub-processes of the command program must terminate before program termination is accepted.

  Failure to observe this dependency hangs the interface until all children exit. Command execution is complete only when it and all processes have terminated.

Additional Information

CPI functions are obtained if you include the file libcpi.h at the beginning of each source file. This is au tomatically included in the pg.template template file. Further information about CPI functions is given in Chapter 6.

If you have developed a program in a different directory from the Command Set directory, and subsequently copied the program to the Command Set directory, the program automatically becomes available to the Command Set. However, debugging is not possible on such a program.

Writing a Program with more than one Sourcefile

If you want to develop a single command comprising more than one source file, then it must be developed in a separate directory and the resulting executable program copied to the Command Set directory.

1. Type the following to create a directory where the program is to be developed:

```
mkdir /hp63000/FE_LIBS/.example
```

2. Type the following to move to the new directory:

```
cd /hp63000/FE_LIBS/.example
```

3. Create a Makefile as follows:

```
mkef
```

4. Add/change the Makefile as follows:

```
CC      = cc
CF      = -I$_EPM_HOMEDIR -MAIN -ET.o
        -I/hp63000/usr/include -I/hp63000/c   lib \
        -I/hp63000/bin/lint -I/hp63000/pmc/ -lmln \
        -I/hp63000   lib
        -I/hp63000   usr/include -I/hp63000/pmc_usr/   \
        -I/hp63000   usr/   lmln -I/hp63000/pmc_usr/l
LD      = /hp63000/
LDFLAGS = -ET.O.    lmln -ET.o.mln -ET./
        -ET.        /   c   -lmln/usr/lib \
        -L          O/
        -L /   /lib   ll

$   = -lcicpc -lfll

T   = <Sourcefile Directory Path where executable program is to be
      depend>O

all ( $(LIBFLAGS) $(LIBS) $(LIBS) $(M_LIBS) -o $(MAIN)O
```

5. Develop your pro as described previously in this section.

6. Type

```
make depend
```

and then

```
make all install
```

Writing a Script

The parameters are passed to the script in the conventional UNIX manner. The parameter count is available through the variable **$#**. Parameters are passed to the script exactly as written to the Command Set and are available in the order specified (left to right).

The following line shows an example:

```
EXEC_INP(init one two three);
```

The script executed is init, and the parameters passed , two, and three as follows:

```
$# = "one two t  ee" where
$1 = "one"
$2 = "two"
$3 = "three"
```

Note that **$0** is 3.

To start a script in background
Command Set call, as shown i

```
EXEC_INP(init one two t
```

Exit Status

The exit status of the script conveys the success or failure of the script execution, and must be set. For a full explanation of the available exit codes, see Chapter 5. The follo g line shows an ple:

```
exit 0;
```

Passing a Comment

To pass a comment to the B n model file variable, write to stdout. All output that you write to st ut is classified as the comment, and is terminated by the termination of the . The length of the comment is set automatically, and is equal to the number of bytes written to stdout.

When a script is started in background mode, comments are ignored.

Direct a Message to the Report W w

To direct a message to the report window from a script, to stderr.

Script Dependencies

When you program a Command Set pt, consider the following dependencies:

- CI_MAX_COMMENT_LEN bytes are written to stdout. This constant is set at 45000 bytes. This value is fixed and cannot be changed.

  Failure to observe this dependency results in a comment truncation warning and possibly causes a script failure with the message ERROR: CI: cli14 Giveb: Write on Pipe. This means that the interface has read the maximum number of bytes allowed and has closed the input medium.

- The script exit status is of one of the following constants:

  CI_CALL_PASS = 0

  CI_TEST_PASS = 0

  CI_TEST_FAIL = 1

  CI_CALL_ERROR = 2

  CI_CALL_BREAKN = 3

  Failure to observe this dependency results in an undefined return status being returned to the production software. This means that zero bytes of comment are stored to the application model file variable. The undefined state is t d in an error and an error message is generated. Manual intervention is required and a RETRY and an ABORT button are made available.

- Any timeout requirements must be satisfied by the user (SIGALRM available).

  Failure to observe this dependency codes an endless loop into the function and results in a production software hangup, except when the NOWAIT option is used. In this case you are responsible for hanging background processes.

- Error Messages are your responsibility, and must be written to stderr. This is the medium for passing messages to the report window.

- If a script forks another process then that process inherits the comment and report passing communication mediums. All grib-processes of the command script must terminate before script termination is accepted.

Failure to observe this dependency hangs the interface until all children exit. Command execution is complete only when it and all child processes have terminated.

4

VEE Command Set

A VEE Command Set contains a VEE instance. This chapter describes how to:

- Install a VEE Command Set
- write a VEE user function

A list of VEE Command Set error messages is given in "VEE Command Set Error Messages" in Appendix A.

Installing a VEE Command Set

To install a VEE Command Set:

1. Type the following at the co d line:

   /hp63000/proc_srv/libin/installt -t VEE

   The installt.srv script es the follo ' to your directory:

   a special Makefile (for debug/nondebug switch)

   a VEE User Function Command Set template ci_vee_obj_tmp

   a VEE Command Set object ex ple file ci_vee_obj_tmp

   a VEE Command Dispatcher (d.v.dispatch)

   a command interface link

2. Create a VEE Command Set as foll (for more details see "Writing a VEE User Function"):

   Start vectest and load ci_vee_obj_tmp.

   Create user objects.

   Convert the objects into User Functions with unique names.

   Save to file ci_vee_obj.

A set of example objects be found in the file ci_vee_obj_exp.

The installation is complete when ci_vee_obj has been created with the desired User Functions for the Command Set. It is this file that is loaded by the interface to make your Functions installable in the production software.

The Installt.srv -t VEE script can be called as often as required and does not destroy any source files you have created. It does, h er, always overwrite the Makefile and the template ci_vee_obj.tmp, so do implement any of your functions in a file with these names.

When you create a User Function you must adhere to the rules as defined in the section "Dependencies".

Writing a VEE User Function

The object declarati for a VEE User Function is as defined in the VEE object template.

The parameters are as follows:

parm_list_input	is a Text terminal. This terminal provides all parameters which are directed at the object, and contains a list of 0 or more tokens that are separated by a space.
return_status	is an integer terminal, and must be set to indicate the success or f of the object execution. For a full explanation of the available codes, see Chapter 5.

Passing a Comment

To pass a comment to the application model file variable, write to stdcmt. All output that you write to stdcmt is classified as the ent, and is terminated by the termination of e object. The le of the comment is set automatically, and is equal to the number of bytes written to stdcmt.

Direct a Message to the Report Window

To direct a message to the report window from an object, write to stderr.

Creating the User Function

When the new object has been created using the mandatory parameters as previously described above, give the object a new unique name and convert it into a User Function using the VEE menu item Make UserFunction. Save the file c1_vee_obj. The new function/object is now av le to the Command Interface.

Dependencies

When you create a VEE Command Set, consider the following dependencies:

- Max CI_MAX_COMMENT_LEN bytes are written to stdout. This constant is set at 45000 bytes. This value is fixed and cannot be changed.

  Failure to observe this dependency results in a comment truncation warning, and possibly a hangup. The interface has read the maximum number of bytes allowed and is ignoring the rest (write with no reader).

- The return_status terminal is assigned one of the following constants:

  CI_CALL_PASS = 0

  CI_TEST_PASS = 0

  CI_TEST_FAIL = 1

  CI_CALL_ERROR = 2

  CI_CALL_THREAD = 3

  Failure to observe this dependency results in an undefined return status being returned to the production software. This means that zero bytes of comment are transferred to the application model file variable. The undefined state is treated as an error and an error message is generated. Manual intervention is required and a RETRY and an ABORT button are made available.

- Any timeout requirements must be satisfied by the user.

  Failure to observe this dependency codes an endless loop into the function and results in a production soft hangup.

- Error Messages are your responsibility, and must be written to stderr. This is the medium for passing messages to the report window.

Additional Information

CPI functions are automatically included in the template ci_vee_obj_tmp. Examples of general CPI functions are provided in the file ci_vee_obj_exp.

When you start a VEE Command Set, a VEE icon appears on the display. Do not kill this icon or you will kill the Command Set.

5

Return Codes

Any unit (function, program, script or VEE object) must return one of the following predefined constants. Where constants are not available (script/VEE), then the respective values [0..3] must be assigned as the return status.

The valid codes are as follows:

CI_CALL_PASS

value = 0

The unit has successfully completed the task and has terminated without error. CI_CALL_PASS is used to indicate that an EXEC_INP, a D_BHDGE, or a PER_END call has been s fully completed,

CI_TEST_PASS

value = 0

Indicates that the test has p
USER_PROC Command Set,

CI_TEST_FAIL

value = 1

Indicates that the test has f

CI_CALL_ERROR

value = 2

A failure occurred during execution of the unit, causing any information written as a comment to be lost. When this constant is returned, NO comment (the empty string) is assigned to the application model the variable and the error state is asserted. You can then select SETW or ASRT.

Note Any value apart from 0 or 1 generates an error popup dialog at the user interface.

CL CALL BREAKD

value = 3

The Break Key has been pressed. This can occur within Test Programming Interface (TPI) functions. If this status is returned the choice of RETRY or ABORT is given (as with an error state).

Any return code apart from the valid range [0..3] generates an error, and is an invalid code. The production software notes the failure, and manual intervention is necessary to rectify the situation. You are given the option of RETRY or ABORT.

6

Command Set Programming Interface

The Command Set Programming Interface (CPI) provides a set of routines for accessing HP SmartTest flags and variables during the execution of a test program, plus a few miscellaneous utilities.

The CPI also includes TPI (Test Programming Interface) functions. See the manual *Programming with C or HP VEE* (12825-91014) for details.

Further information about the functions described in this chapter is given in Appendix C which contains the file

```
/opt8000/prod_com/include/libcipi.h
```

Flag Classifications

Production Flags

Production flags typically have a boolean 0/1 value representing the states on and off. However, some of the flags contain an integer value (for example, OX_sample_size).

The Production Flags each have a PREDEFINED name, and are grouped into the following categories:

- Testbase Flags
- TestFlow Flags
- System Flags

All production flags are declared as type i

User Flags

User Flags are defined in the Graphical TestFlow Editor variables page, and take on the name that the user specifies. User flags are of type Integer.

User Variables

User Variables double or string can be declared in two ways:

1. explicitly in the Variables page of the TestFlow Editor
2. implicitly in the testflow itself

ModelFile Variables

ModelFile Variables are implicitly declared in the Application Model File. They are of the type string.

Access Routines

The Command Set Programming Interface provides a variety of functions to get and set the Flags and Variables as previously described. In the Production set routines, the value passed or by the case is validated to ensure that the range of the flag is not exceeded. For some flags, the unvalidated value is passed on to the flag.

The routines are grouped into get/set functions for TestState, TestFlow, System, User and Modifiable flags/variables. Where a flag is set, the parameter description is named Flag. For a variable, the parameter description is name. When Tasmane occurs in the function get/set TestStateFlag, it is a reference to the testcase where the flag is located.

Although all flags are internally defined as an int, the routines declare a Long. This is simply for compatibility reasons with VIX which does not have an int type. The interface is robust because you cannot exceed flag limits on set, and the get routines internally typecast the int to a Long. Because the underscore character is illegal within VIX functions, they have been omitted within function and parameter names.

Some get, and set functions are specific to multi-site test. Unless the function contains a parameter that allows you to specify a site number (for example GetSystemFlagSiteIn), always assume that the function refers to the site that is currently in focus. For further details about when a site is in focus, see the manual *Developing a Production Test Program* (Part No. E3024-01011). When you specify a site in one of these functions, the site number must always be less than or equal to the number of sites defined by CI_CFG_MAX_SITES.

All functions are type dependent, so it is not possible to release the function SetUserFlag (for example) to set a SystemFlag and vice versa. Each routine has a clearly defined purpose in the CPI set.

Production Flags

get

```
long GetStatusFlag(char *tasname, char *flag, long *value)
long GetStatusFlagOfSite(int *tasnre, char *flag, long site, long *value)
long GetTestFlag(char *flag, long *value)
long GetSystemFlag(char *flag, long *value)
```

```
long GetSystemFlagOfSite(char *      long site, long *value)
```

Note that you can use the function GetTestsuiteName described later in this chapter to find out in which testsuite a user procedure was executed.

set

```
long SetTestsuiteFlag(char *tsname, char *flag, long value)
long SetTestsuiteFlagOfSite(char *tsname, char *flag, long site,
long SetRunflowFlag(char *flag, long value)
long SetSystemFlag(char *flag, long value)
long SetSystemFlagOfSite(char *flag, long site, l    *v
```

User Flags

get

```
l        UserFlag(char *flag, long *value)
,        UserFlagOfSite(char *flag, long site, l    *v
```

set

```
long SetUserFlag(char *flag, long value)
long SetUserFlagOfSite(char *flag, lo    lo    *v
```

User Variables

get

```
long GetUserDouble(char *name, double *value)
long GetUserDoubleOfSite(char *name, long site,        e *      )
long GetUserString(char *name, char *value)
long GetUserStringOfSite(char *name, long site, char *value)
```

set

```
long SetUserDouble(char *name, double value)
long SetUserDoubleOfSite(char *name, long site, double *v    e)
```

```
long SetUserString(char *name, char *v
long SetUserStringOfSize(char *name, lo
```

ModelFile Variables

get

```
long GetModelfileString(char *name, char *v   c)
```

set

```
long SetModelfileString(char *
```

Type Analysis

If you do not know the type of a user you can find this out using the function GetUserType. This function sets the parameter type to one of the following three constants:

```
CI_CPI_FLAG = 0 /* flag */
CI_CPI_DBL = 1 /* double */
CI_CPI_STR = 2 /* string */
long GetUserType(char *name, long *type)
```

Return Values

All return values of the functions assume one of three values which are consistent over all routines:

get 0 = ok, 1 = error on name or flag (type, -1 = internal error

set 0 = ok, 1 = error on name/type/value, -1 = internal error

0: The routine get/set without error

1: The name for the flag or variable is invalid because, the type is inconsistent with the function used (for example, setting a double variable with SetUserFlag), or the value presented to the function exceeds the limits of the flag.

-1: An internal error has occurred (bug condition)

Flag Access

This section describes the flags available through CPI routines, and the various methods in which they are specified.

Direct String

Use the quoted string method to specify e to querybet:

```
long val;
GetSystemFlag("dataLog_to_file", &val);
```

Predefined Strings

A set of constants to specify flag names are available. For example:

```
long val;
GetSyst   1  (CI_DATALOG_TO_FILE, &val);
```

Flag List

The following constants specify flag names.

GetTestsuiteFlag

CI_BYPASS

CI_SET_PASS

CI_SET_FAIL

CI_HOLD

CI_HOLD_ON_FAIL

CI_OUTPUT_ON_PASS

CI_OUTPUT_ON_FAIL

CI_VALUE_ON_PASS

*CI_VALUE_ON_FAIL

CI_PER_FTK_ON_PASS

CI_PER_FTK_ON_FAIL

CI_FORCE_SERIAL

CI_EXECUTED

CI_PASSED

CI_FPC_ON_FAIL

CI_FPC_ENABLE

GetTestShowFlag

CI_SET_PASS_LEVEL

CI_SET_FAIL_LEVEL

CI_SET_BYPASS_LEVEL

CI_PAUSE_ON_FAIL

CI_OPTIMIZED_MODE

CI_GLOBAL_HOLD

CL_DEBUG_MODE

CT_GLOBAL_OVERON

CT_GLOBAL_PARALLEL

GetSystemFlag

Flags with Read & Write Permissions

CL_REPORT_TO_FILE

CL_REPORT_TO_PRINTER

CL_DATALOG_TO_FILE

CL_DATALOG_TO_PRINTER

CL_DATALOG_TO_REPORT_WINDOW

CL_DATALOG_FORMATTER

CL_DATALOG_SAMPLE_SIZE

CL_GRAPHIC_RESULT_DISPLAY

CT_STATE_DISPLAY

CL_PRINT_WAFERMAP

CL_OOC_WATCH

CL_OOC_SAMPLE_SIZE

CL_INK_WAFER

CL_MAX_REPROBES

Flags with Read Permissions

CL_ABORT

CL_EXIT_TESTFLOW

CL_RESET

CL_PAUSE

CL_REFERENCE

CL_CHECK_DEV

CL.RETEST

CL.RETEST..W

CL.SKIP

CL.QUIT

CL.OFFLINE

CL.GOOD..PART *site specific*

CL.BAD..PART *site specific*

CL.BIN..NUMBER *site specific*

CL.BIN..CODE *site specific*

CL.LEVEL..END

CL.TEMP..MONITOR

CL.CALIB..AGE..MONITOR

CL.DIAG..MONITOR

CL.CURRENT..MONITOR

CL.REPHOBE..E R

CL.DEV..E

CL.DEV..HOLD

CL.OOC..WARN

CL.OOC..STOP

CL.IDLE

CL.SITE..SET-UP *site specific*

CL.SITE..REPROBE *site specific*

Miscellaneous

The following functions are included in the Command Set Programming Interface and are independent of flags or variables.

Die Positions

get

Retrieves the currently set x and y die coordinates of the die about to be tested, or which has just been tested (depending on whether the call is made before or after a device test. The parameters x and y are defined as longs for compatibility with VEE, but are actually return values that can fit into an int.

If multiple sites are used, the second function (GetDiePosATSite) allows you to retrieve die coordinates from a specific site. If you use the first function (GetDiePosAT), the die coordinates for the site currently in focus are returned.

```
long GetDiePosAtSite  *x, long *y)
long GetDiePosATSite(long site, long *x, long *y)
```

Sets the x and y die coordinates of the die about to be tested when used before a device test takes place. The parameters x and y are defined as longs for compatibility with VEE, but are internally typecast to an int.

The second function (SetDiePosATSite) allows you to the die coordinates for a specific site. If you use the first function (SetDiePosAT), the die coordinates for the site currently in focus are set.

```
long SetDiePosAT(long x, l    y)
long SetDiePosATSite(l    site, l    x, lo
```

Temperature

get

Retrieves the string currently set for temperature and copies it to the string pointer *temp.

This function writes a maximum CI_GPI_MAX_TEMP_LEN + 1 bytes to the pointer *temp.

```
long GetTemperature(char *t p)
```

set

Sets the string recorded as device test temperature to the value pointed to by string pointer *temp.

```
long SetTemperature(char *t )
```

Package ID

get

Retrieves the string currently set as the Package Id and copies it to the string pointer *id.

This function writes a maximum CI_GPI_MAX_PACKAGE_LEN + 1 bytes to the pointer *id.

If multiple sites are used, the second function (SetPackageIdOfSite) allows you to retrieve the information from a specific site. If you use the first function (GetPackageId), the information for the site currently in focus is returned.

```
long GetPackageId(char *id)
long GetPackageIdOfSite(long site, char *id)
```

set

Sets the string recorded as device test Package Id to the value pointed to by string pointer *id.

The second function (SetPackageIdOfSite) allows you to set the information for a specific site. If you use the first (SetPackageId), the information for the site currently in is set.

```
long SetPackageId(char *id)
long SetPackageIdOfSite(l site, *id)
```

ult File

get

Retrieves the file name (with the software extension) that logs the results for level *lo .

```
1     GetResultFile(     *lowname, char *value)

      e
```

get

Retrieves the Wafer Description file name that is being used to generate x and y die test coordinates.

This function writes a maximum CI_CFI_ _PATH_LEN + 1 bytes to the pointer *path.

```
long  GetWaferDescriptionFile(char *path)
```

Generating Popups

Popup Message

This function generates a message popup. The popup appears with two default buttons and the message in *msg. When any parameters *b2a to *b7a are non-empty strings, additional buttons between quit and cont also appear (see Figure 6-1).

```
long  MsgDisplayMsgGetResponse(char *msg, char *b2a, char *b3a,
      char *b4a, char *b5a, char *b6a,      *b7a,
      long *response)
```

Figure 6-1. Popup Message

The popup disappears when any button is pressed and the parameter *response is set to the button number selected as follows:

```
*response = 1;  quit
*response = 2;  user button 2 (*o2a)
*response = 3;  user button 3 (* a)
*response = 4;  user button 4 (* a)
*response = 5;  user button 5 (*o5b)
*response = 6;  user button 6 (*o6a)
*response = 7;  user button 7 (*o7a)
*response = 8;  cont
```

Button strings are restricted to a maximum of CI_MAX_BUTTON_STR_LEN = 6. The Message string is limited to a maximum of CI_MAX_MESSGE_STR_LEN = 512.

Popup Message with Input

This function generates a message popup with a string input field (Figure 6-2), and is otherwise similar to MsgDisplayMsgGetStringResponse.

```
long  MsgDisplayMsgGetStringResponse(char *msg, char *b ,
      char *b5b, char *b6b, char *b6c, char *b6d,
      char *b7a, long *resp  e, char *retring)
```

Figure 6-2. Popup with Dialog

The pops when any button is pressed and the parameter *response is set to the b n number selected. If RETURN is used to terminate the string input, the popup also disap s, and *response is set to -1.

Button strings are restricted to a length of CI_MAX_BUTTON_STR_LEN = 6. The Message string is limi to the 1 CI_MAX_MESSGE_STR_LEN = 512. The string is written to the pointer *string, and may be up to CI_MAX_MESSGE_STR_LEN + 1 (513) bytes long.

Command Set Message File

NlsGetMsg is a function that allows you to define and use a local catalog file as part of a Command Set. When you use it within a program, messages may be changed or modified without further code changes.

```
char *NlsGetMsg(long la    )
```

Example

1. Create the messages/string lines in any text file using the format as described by the gencat UNIX command:

   file mfile.msg

   ```
   2001  this is message 2001
   2002  this is message 2002
   ```

2. Generate the m file the command:

   ```
   /tp2000/prod_aix/bin/install_cs -m mfile.msg
   ```

3. Messages in the catalog file may now be accessed with the NlsGetMsg routine, for example:

```
fprintf(stderr, "%s", MsgGetMsg(2001));
```

generates the ` ` this is message 2001 on the report window.

You might also find it useful to refer to the HP-UX man pages for *mkcapes*, *cmkcaps* and *catget*.

Disable Error Messages

This function disables the generation of error messages from the CPI to the Report Window.

```
long DisableErrorMessages()
```

Enable Error Messages

This function allows error messages to be sent f CPI to the Report Window. Use the EnableErrorMessages fun y if error message reporting has previously been ` led using the dis e error messages function.

```
long EnableErrorMessages()
```

Retrieve the User Mode

The GetUserMode function retrieves the current on from the HPSecurity software.

```
long GetUser  e(long *mode)
```

The long pointer a can be one of the following:

```
CI_CONSOLE_MODE=1, for Console mode

CI_OPERATING_MODE=2 for Operating mode

CI_SUPERVISOR  3_MOE  for Supervising mode

CI_ENGINEERING_MOD  for Engineering mode
```

If an error occurs, *mode* is undefined.

Set the Operator Console Default Message

The SetIdleMsg function sets the default message to be displayed on the Operator Console when no is being performed.

```
long SetIdleMsg(char *line0, char *line1)
```

*line0 can be up to 20 characters long, and *line1 can be up to 19 characters long.

Execute a TestFlow

This function executes the TestFlow currently loaded in the Data Manager. It performs the same functions as the executa all option in the TestFlow Editor.

```
long ExecTestFlow()
```

Waferman Result Display Dimensions

get

- function retrieves the wafer dimensions of the waferman test result
- using

The six values defined in the function represent a two dimensional coordinate system for the dice on a wafer as shown in Figure 6-3. The parameters min_x, max_x, min_y and max_y define the range of the X values or Y values. The parameter quadrant (ranging from 1 through 4) defines the direction of increasing values.

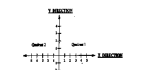


Figure 6-3. Waferman Dimensions

The parameter orientation is the angle of the flat wafer site (an angle of 0 means that the flat is at the bo).

The returned values are only valid if a wafer has been tested, otherwise the results are undefined (for example during package testing).

```
long GetWafer0    i  (long *min_x, long max_x, long *min_y,
                      long max_y, long equadrant,
                      long orientation);
```

out

This function sets new dimensions of the waferman test result display. The meaning of the six values are identical to those described previously for GetWaferDimensions, and as shown in Figure 9-5.

The new coordinate system defined with SetWaferDimensions comes into effect with the next display operation.

```
long SetWaferDimensions(long min_x, long max_x, long min_y,
                        long max_y, long quadrant, long orientation);
```

Retrieve the Testsuite Name

This function returns the name of the te ite that contains the current user procedure. The pointer testsuitename points to a user-provided area to which the testsuite name is copied. The area must be able to hold at least CI_CFG_MAX_TESTSUITE_LEN characters.

```
long GetTestsuite0    (char *testsuitename)
```

The following example how this function is used:

```
char current_testsuite        [CI_CFG_MAX_TESTSUITE_LEN];
.
.
.
/*  a testsuite name is printed in   Report Win   */
Ret0  testsuiteName(current_testsuite   );
printf(sprintf(ret0ret0, "Current testsuite is %s\n", current_testsuite_name);
```

Testunite Flag

get

GetTestuniteFlag retrieves the current value of the (#Flag) in a given testunit(#testunit) and deposits it in long, pointed to by #value. Flags are actually declared as ints, but are typecast to longs within this function for compatibility with V

GetTestuniteFlagInts iets you fy which sits should be used (1,2, . . .). while GetTestuniteFlag numerics the sits currently in focus (for example during sequential User Procedure execution). In case of ' site test the first sits is always in focus.

```
long GetTestuniteFlag(char #testunit, char #flag, long #value)
long GetTestuniteFlagInts(char #testunit, char #flag, long #ints, long #value)
```

The flag may be assigned one of the following constants:

- CI_BYPASS ..... : R/W
- CI_SET_PASS ..... : R/W
- CI_SET_FAIL ..... : R/W
- CI_HOLD ..... : R/W
- CI_NOGO_ON_F ..... : R/W
- CI_OUTPUT_ON_PA ..... : R/W
- CI_OUTPUT_ON_F ..... :
- CI_VALUE_ON_P ..... :
- CI_VALUE_ON_FAIL ..... :
- CI_PER_FN_ON_PASS ..... :
- CI_PER_FN_ON_FAIL ..... :
- CI_FORCE_CRITICAL ..... :
- CI_EXECUTED ..... :
- CI_PASSED ..... :
- CI_FPC_ON_FAIL ..... ie for PS60)
- CI_FPC_ENABLE ..... ie for PS60)

Alternatively, use a " " ma " ed. A "string" is used to specify the TestUnite

The values returned are as f

0 ok

1 error on a name

-1 internal error

The follo ple shows how this f

```
void CMainpsiDoc
    {
    char
    char
    int
    int
    int
```

```
long  GetUserFlag(char *fl  , long *value)
long  GetUserFlagIfSite(    *flag, long site, l    *value)
```

e values returned are as foll &

0 ok

1 error on a name

-1 internal error

e following example shows how this f

```
void CsRemoveSite(char *name,int,int,int,
    int  permission;
    int  *numbers[10];
    int  *numbers;
    int  No_set;
{
  for i= numbers i    in to
                  i
                  i
                  i
```

set

SetUserFlag sets the value of the user defined flag #flag with the value value. The flag specified must have been declared as a userflag. Flags are actually declared as ints, but are defined as longs within this function for compatibility with VEE.

SetUserFlagSite lets you specify which site should be used (1,2, ...), while SetUserFlag assumes the site currently in f (for example during sequential User Procedure execut) in case of single site test the first site is always in focus.

Use a "string" constant to identify the flag req t.

User Double

get

GetUserDouble retrieves the current value of the user defined variable, *name, and deposits it in the double, ed to by *value. The variable must have been declared as a ble.

GetUserDoubleSite lets you specify which site should be used (1,2, ...), while GetUserDouble assumes the site currently focus (for example during sequential User Procedure execution). In case of t site test the first site is always in focus. Use a "string" co t to identify the le required.

```
long  GetUserDouble(char *name, double *value)
long  GetUserDoubleSite(char *name, l  site,  le *value)
```

The values ed are as follows,

0 ess

1 error on a name

-1 internal error

The example for *User Flag* shown how this function is

set

SetUserDouble sets the value of the user defined v le, *name with value. The flag specified must have been declared as a double.

SetUserDoubleSite lets you specify which site should be used (1,2, ...), while SetUserDouble assumes the site currently in focus (for example during

sequential User Procedure execution). In test the first site is always in focus.

Use a "string" constant to identify e

User String

get

GetUserString retrieves the current value of the user defined variable, *name, and copies it to the string pointed to by *value. The v le must have been defined as a string.

GetUserStr OfSite lets you specify which site should be used (1,2, ...), while GetUserString assumes the site currently in focus (for example during sequential User Procedure execution). In case of single site test the first site is always in focus. Use a "string" constant to identify the variable required.

This function writes a maximum CI_GPI_ X_USER_STRING_LEN + 1 bytes to the pointer *value.

```
long GetUserString(char *name, char *value)
* long GetUserStringOfSite(char *name, i site, char *value)
```

The values returned are as follows,

0 ok

1 error on a name/type

-1 internal error

The example for *User Flag* shows how this function is used.

set

SetUserString sets the value of the user defined variable, *name with the string value.

SetUserStringOfSite lets you specify which site should be used (1,2, ...), while SetUserString assumes the site currently in focus (for example during sequential User Procedure execution). In case of single site test the first site is always in focus.

Use a "string" constant to identify the variable required.

Determine First Use of a User Procedure

This function can be used during testing of multiple sites. In order to write user procedures that operate in parallel on all sites, a user procedure must be able to find out if the current call to the user procedure was the first one in the context of a testsuite execution. It could then do all operations on the first call and only return values when invoked afterwards.

This function re a 1 the first time that the user procedure makes a call to any site within the t site. Any subsequent calls of the user procedure within the testsuite return a 0.

```
long IsFirstInvocation(long site_first)
```

Determine the Sites to be Addressed

This function is used during testing of multiple sites to determine which sites should be addressed by a user procedure. It fills a caller-provided list of integers with the site numbers that the user procedure should operate on. A site number must be greater than 0, and a 0 marks the end of the list.

Note that the list must have a minimum CI_CFI_MAX_S +1.

```
long GetSitesToBeTested(int *sit _to_be_d )
```

The following lines show an example of this function:

```
int to_do[CI_CFI_   _SITES+1], i;
.
.
.
if ( it T wTest (to_do) == 1)
for (i ; to_ [i] > 0; i++)
  _s  tk'  _with_site(to_ [i]);
  /* to_do[i] is a site n  ber */
```

7

Debugging

A prerequisite for this chapter is that you are familiar with:

- the *adb* function
- the *make* command
- and have a good understanding of UNIX.

Function Debug

This section describes how to debug a Function Command set.

A debug can only be performed when the Command Interface can load the shared library libcliset.sl. Note that a list of all error messages associated with the Function Command Set is given in "Function Command Set Error Messages" in Appendix A.

Preparing the Debug Session

1. Type make debug at the command line. This prepares the debug environment.
2. Start the production software. This causes an xdb window to appear for debugging the Command Set.
3. Set a break point on the functions you want to debug, and type C for continue. You cannot see the source code for the interface itself, but when a breakpoint is reached, you automatically see the source code for your functions.

Debugging

See the xdb page in the UNIX manual. Standard xdb debug rules apply.

Ending the Debug Session

Release all breakpoints and type C to run the Test Program to completion.

7 Leave xdb and type the command make clobber all to remove all files/install the normal version of the Interface. Modify or fix any problem found during the debug session.

Program Debug

To debug one or more programs, first compile each one with the debug option. Although the Command Set may use a large set of programs, the Command Interface allows you to interactively debug any number from one to all of the them.

A full list of Program Command Set error messages is given in "Program Command Set Error Messages" in Appendix A.

Preparing the Debug Session

1. Type install_debug prog1 prog2 . . . progn at the command line. Progl to Progn is a list of all the programs you want to debug. This action:

   touches each prog .c

   creates a prog .d version of each program

   links in the debug version of the interface

   and performs a make debug.

2. Start the production software.

Wherever possible, the Command Interface starts the prog .d version of the program. The prog .d version is a script which generates an hpterm and calls the xdb debugger on the program. Because xdb stops the program, the effect is that the production software stops when the program is called. When the debug session on the program is finished, the production software continues. TestProgram execution until the next debug program is encountered (if any).

Using xdb, it is not possible to specify startup parameters when you start a program. If parameters are to be supplied to the program, you must set them manually within xdb.

Debugging

See the xdb page in the UNIX manual. Standard xdb debug rules apply.

Ending the Debug Session

Leave xdb and for the TestProgram to terminate or abort it. Type the command make clobber all to recompile/install the normal version of the interface. Modify or fix any problem found during the debug session.

Script Debug

You can use the normal UNIX facilities to debug a script. These may be debugged on-line at a shell prompt.

VEE Debug

Debugging the VEE interface is done interactively by set ' brnakpoints on the user objects to be debugged. Before that, you must a debug version on the Command Interface.

Note that a full list of VEE Command Set error me in "VEE Command Set Error Messages" in Appendix A.

Preparing the Debug Session

1. Type make debug at the command line.

, generates a link to ci_process.d

copies in the debug version of ci_v_dispatch (as opposed to the normal dispatcher).

2. Start the production software.

When the Command Set is loaded into the DataManager, a VEE screen appears with the Command Interface Debug module automatically loaded. You must:

manually merge ci_vee.obj (so that user objects can be deb)

press the [START] button.

The interface now receives commands in the normal fashion, t may now be interrupted by the user.

Debugging

Each command received by the interface can be seen in a special COMMAND object. A breakpoint is already set on the object that receives command requests (vpx). Set a breakpoint on an object embedded within the UserFunction that you want to debug. When the commands are received they are executed, unless a breakpoint is set.

Debug objects as they reach their bre points. Use the VEE debug techniques as described in the manual *Using HP VEE: ne and HP VEE/Test*.

Ending the Debug Session

Leave VEE and type the command make clobber all to recompile/install the normal version of the interface. Modify or fix any lem found during the debug session.

8

Software Updates

This chapter contains information needed if you make an update to the software.

Software Updates

After any software updates on the Command Interface, call:

```
make_all_ci_processes
```

Use this command if a CI sourcefile or the general purpose library has been modified. Call `make clobber all | tail` on all CI/bin/libs/ci and CI/bin/libs/fci. do update and install the new executables.

or

```
make_all_ci_u
```

Use this command when the programming interface (CPI) has been modified. Use `make clobber all` on every user Command Set where a makefile exists.

Note that after a software update, all Command Sets belong to root. Therefore, after the update, use chown or chgrp as required to change the owner or group.

Command Set Dependencies

Note the following:

- All Poseidon Command Sets are dependent on `libcicpi.a1` (CPI functionality).
- All Program Command Sets are dependent on `libcicpi.a1` (CPI functionality).
- All VEE Command Sets are dependent on `ci_vee_obj` (which contains CPI functionality).
- All Ci processes are deposited in `prod_usr/lib`, and are accessed through the use of links from Command Set directories. If any Ci is updated, the update is automatically seen by every user Command Set, without intervention.

Ci Dependencies

All command interface processes are dependent on `libci.a1`, the Ci general purpose library.

A

Error Messages

This appendix contains the error messages associated with a Function Command Set, a Program Command Set, and a VEE Command Set.

Function Command Set Error Messages

The following are the error messages if an error occurs when the Command Set is used:

```
CI: ci_load_lib: illegal .sl:
CI: file not a lib!
```

This error occurs when the file *libclject.sl* is not a shared library. Solve the problem by using the Make command to cr a shared library.

```
CI: ci_load_lib: illegal .sl:
CI: undefined s (s):
```

This error occurs when the file *libclject.sl* contains undefined symbols, that are not permitted. A list of the undefined symbols is generated in the Report Window. Solve the problem by ensuring that all function names are spelled correctly, and by ensuring that all functions are located in the correct library.

```
CI: ci_load_lib: illegal .sl:
CI: handle/index error
```

This error occurs when an inconsistency exists between the Command Interface process and the shared library. Remake the library and start again.

```
CI: ci_load_lib: ill sl .sl:
CI: address space ??
```

This error occurs if not enough address space exists to load the shared library. This error can only be resolved by the System Administrator.

```
CI: ci_ld  _lib: illegal .sl:
CI: doesn't exist
```

The error occurs if no shared library exists. Make and generate the shared library.

```
CI: ci_load_lib: illegal .sl:
CI: permission ??
```

Change the access permission to the shared library.

The following warning can occur under general execution of a function command set:

```
CI: execute_command: comment truncation warning
```

This message occurs when too many bytes have been included in the comment, causing it to be truncated. Solve the problem by looking at the command that you have implemented and reducing the number of bytes written in comment.

The following are general internal processor errors that can only be resolved by HP personnel:

```
CI: gac address: symbol table overflow
```

This warning occurs when an overflow occurs during the storing of function symbols. This is a non-fatal situation. The Command Set may become slower in operation due to having to fetch function addresses at every call.

```
CI: ci_load_lib: internal error '
CI: ci_load_lib: illegal message queue id
CI: ci_load_lib: illegal action identifier

CI: s  ste_command: illegal message queue id
  : c      ck: illegal message queue id
```

A

Program Command Set Error Messages

The following error messages can occur when you execute a program Command Set:

```
CI: execute_command: command t     tion warning
```

This message occurs when too many bytes have been included in the command, causing it to be truncated. Solve the problem by looking at the command that you have implemented and reducing the number of bytes written as command.

When a program crashes, the following message is displayed:

```
CI: child died:<re    >
```

where reason is one of the f ing:

```
"",
"Q    ",
"Data     ",
" ia",
"Illegal Instructi
"Trace Trap",
"IOT Instruction",
"EMT Instruction",
"Floating Point Exception",
"Kill",
"Bus Error",
"Segmentati  Violation",
"Bad arg to syst  call",
"Write on Pipe",
"Alarm Cl.  ",
"Terminate Signal",
"User Signal 1",
"User Signal 2",
"Death of Child",
"Power Fail",
};
```

or when the software cannot identify the

```
CI: child died: signal <num>
```

where num is the signal number r vod.

Solve this error by debugging the program and then fixing the error.

The following are general internal program errors that can only be resolved by HP personnel:

```
CI: ci_main: illegal mess a queue id
CI: ci_main: illegal action identifier

CI: execute_command: illegal message queue id
CI: cons_check: illegal message queue id
```

VEE Command Set Error Messages

The following error message can occur when you execute a VEE Command Set:

```
CI: execute_command: command truncation warning
```

This message occurs when too many bytes have been included in the command, causing it to be truncated. Solve the problem by looking at the command that you have implemented and reducing the number of bytes written as command.

The following are general internal program errors that can only be resolved by HP personnel:

```
CI: ci_main: illegal message queue id
CI: ci_main: illegal action identifier
CI: execute_command: illegal message queue id
CI: run_check: illegal message queue id
```

B

Templates

This appendix contains the template files for a function, a program, and a script.

Function Template File

The following file is an example of the template file *fn.templ.c*:

```
/*****************************************************************
*$Id
*$Id                    MODULE  : fn_templ.c
*$Id
*$Id-----------------------------------------------------------------
*$Id
*$Id          (c) Copyright HEWLETT-PACKARD COMPANY, Bobbl      1993
*$Id
*$Id-----------------------------------------------------------------
*$Id
*$Id  AUTHOR  : Eric Furnmond, HTD-SMO
*$Id
*$Id  DATE    : 17.09.92
*$Id
*$Id-----------------------------------------------------------------
*$Id
*$Id  COMMENTS: T  hsis for user writt  functions.
*$Id
*$Id
*$Id  Instructions:
*$Id
*$Id  1) Copy this template to an appr .c files   you r
*$Id
*$Id  2) Use the command 'make depend' to    ilds the new
*$Id     source files to the makefile utility
*$Id
*$Id  3) Run the command 'make' to generate libclient.sl - the
*$Id     shared library needed for the command interface.
*$Id
```

```
*XXX
*XXX++++++++++++++++++++++++++++++

/*
 *-- system incl  ----------------
 */

        ctmfc.

/*
 *-- ex        1
 */

#include "qsi_c.h"
#include "ti_types.h"
#  #include "timcsys.h"

/*
 *-- external functions ------------------------------------------
 */

/*
 *-- external variables ------------------------------------------
 */

/*
 *--   fines ------------------------------------------------------
 */

#define MIN(MMM,n,h)                          ((n) < (h) ? (n) : (h))

/*
 *-- typedefs ----------------------------------------------------
 */

/*
 *-- global variables --------------------------------------------
 */

/*
 *-- I    lons ------------------------------------------------------
 */

/XXX++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*XXX
*XXX  FUNCTION NAME: general
*XXX
*XXX  DESCRIPTION:
*XXX
```

```
*XII This root          is of how a CommandSet call as specified in
*XII the appl           file can be realized as a user function. The
*XII function it  f is a tripleX  m  is of string manipulation, but it
*XII does demonstr      of all        t and output parameters
*XII and their str      h limitati
*XII
*XII ------------------------------------------------------------------
*XII
*XII Valid Return Co
*XII
*XII CI_CALL_PASS    = 0  f          completed succe
*XII CI_TEST_PASS    = 0  test    end
*XII CI_TEST_FAIL    = 1  test failed
*XII CI_CALL_ERROR   = 2  function call failed
*XII CI_CALL_ERROR   = 3  break
*XII
*XII Constants used:
*XII
*XII CI_CI_MAX_COMMENT_LIN = 65000 bytes
*XII
*XII ------------------------------------------------------------------
*XII Retrieves the input parameter str'      if not     the result on the
*XII value to be assigned to the application model file v  iable
*XII
*XII Function Call:
*XII
*XII eval = EXPS_INP(varnumd abl);
*XII
*XII Result: eval is assigned the value '0' ')
*XII
*XII ------------------------------------------------------------------
*XII
*XII Very Important: always restrict the      th of comment_out. If strnum
*XII is used then a terminating '0x' is not  strictly  placed at the end
*XII of the string, even if the full CI       _LEN bytes are written
*XII
*XII ------------------------------------------------------------------
*XII INPUTS: char *param_list,   *   * parameter string pointer
*XII         int  parameter      No. of tokens in param_list_input
*XII
*XII OUTPUTS: char *comment_out      output/result buffer
*XII         int *comlen             length (in bytes) of comment data
*XII         int *status_out         success state of the function call
*XII
*XII ------------------------------------------------------------------
```

```
*
*
* WRITTEN BY: Eric Paramour, ITD-R&D    DATE: 1993
```

```
*
*
*****************************************
void reversl (
        char  *pcmd_list_input,
        int   pcmnount,
        char  *comment_out,
        int   *comlen,
        int   *stsda,
    )
{
    char *p;
    int i;

    if (      cmnt    C)
    {
        /* Very Important:                  : the length of comment_out */

        strcpy (comment, s, ""
        s    sfcomment,   ,"                  s", CI_MAX_COMMENT_LEN);
    }
    else
    {
        p = comment_out;                     /* set pointer to result buffer */

        /* The maximum :                 )      here is 78 bytes, so it is safe
        * to make a direct c            )      pointers without checking if we
        * exceed CC    ,C              _      bytes
        */
        for (i = strl          _list,   ei) -i; i >= 0; i--) /* reverse copy */
            *(   ) =                  i[i];

                ;                            /* make sure string is terminated */
    }

    if(st     , "FUNCTIIN: report window angle");

        = strlen(comment_out);
    . t = CI_CALL_PUSH;
```

Program Template File

The following file is an example of the template file *pg_templ.t*:

```
/*ZZZ*****************************************************************
*ZZZ                                                                  *
*ZZZ                        MODULE : pg_templ.c                       *
*ZZZ                                                                  *
*ZZZ*****************************************************************
*ZZZ                                                                  *
*ZZZ                 (c) Copyright HEWLETT-PACKARD COMPANY, Bcelona 1983
*ZZZ                                                                  *
*ZZZ*****************************************************************
*ZZZ                                                                  *
*ZZZ  AUTHOR : Eric      ek, ZZD-PAO                                  *
*ZZZ                                                                  *
*ZZZ  DATE   : 30.05.83                                               *
*ZZZ                                                                  *
*ZZZ*****************************************************************
*ZZZ                                                                  *
*ZZZ  CONTENTS: Template for user written programs.                   *
*ZZZ                                                                  *
*ZZZ*****************************************************************
*ZZZ                                                                  *
*ZZZ  Instructions:                                                   *
*ZZZ                                                                  *
*ZZZ  1) Copy this te    ate to ur samp .c files as you               *
*ZZZ                                                                  *
*ZZZ  2) Use the c       depend' to   e visible                       *
*ZZZ      ource files to    makefile utility                           *
*ZZZ                                                                  *
*ZZZ  3) Use the c       ' to gr   ate executables                    *
*ZZZ                                                                  *
*ZZZ*****************************************************************
*ZZZ  DESCRIPTION:                                                    *
*ZZZ                                                                  *
*ZZZ  This program is an example of how a command-cut call as specified in
*ZZZ  the application model file can be realized. The program itself is a
*ZZZ  trivial example of string manipulation, but it does demonstrate usage
*ZZZ  of the input and output medium para    for programs
*ZZZ*****************************************************************
*ZZZ                                                                  *
*ZZZ  Valid Exit Codes:                                               *
*ZZZ                                                                  *
*ZZZ  CC_FAIL_PARS   = 1  program completed successfully              *
*ZZZ  CC_TEST_PASS   = 2  test passed                                 *
*ZZZ  CC_TEST_FAIL   = 1  test failed                                 *
*ZZZ  CC_FAIL_SEARCH = 2  program failure                             *
*ZZZ  CC_FAIL_BREAKS = 3  kpi break                                   *
*ZZZ                                                                  *
*ZZZ*****************************************************************
```

```
*121  8     a the input parameter string and prints the result to stdout
*121  as the value to be assigned to the application model file variable
*121
*121  Program Call:
*121
*121  *val = EXEC,   (resume) abc);
*121
*121  Result: *val is assigned the value 'abc'
*121
*121  ------------------------------------------------------------
*121  INPUTS:  char *argv[]        input parameter array
*121           int argc            No. of parameters
*121
*121  OUTPUTS: stdout              output/result buffer
*121           stderr              directed to report window
*121           exit() code         success state of the program call
*121
*121  ------------------------------------------------------------
*****************************************************************/

/*
*-- system include ------------------------------------------------
*/

#include <stdio.h>

/*
*-- module include ------------------------------------------------
*/

#include "gci_c.h"
#include "ci_types.h"
#include "libcicpi.h"

/*
*-- external functions --------------------------------------------
*/

/*
*--     al variables -------------------------------------------
*/

/*
*--  Pines ---------------------------------------------------------
*/

#define MIN(MMM(a,b) .                                  ) < (b) ? (a) : (b)

/*
```

```
 *-- 1
 */

/*
 *--
 */

/*
 *-- functions ------------------------------------------  ------------------------------
 */

/*****************************************************************  ****************/
/*****************************************************************  ****************/

main(argc, argv)
int argc;
char *argv[];
{
    /* local v
      i 1;

       C);

    if (     == 1)
    {
            tf("          wrr");
    }
    else
    {
      /* The maximum length of all parameters is 70 bytes, so it is safe to
       * make a direct printf without checking if we exceed
       * CI_MAX_CURRENT_LIN bytes
       */

      for(k = (    c - 1); k > 0; k--)    /* step back through arguments */
      {
            printf( s",          );
            printf(" ");
      }
    }

    fprintf(s      , "PROGRAM: report v"     ver\n");

    exit(          );                      /* program completed without error */
}
```

Script Template File

The following file is an example of the template file sc_templ.c:

```
#! /bin/sh
#------------------------------------------------------------------------
#  Copyright (c) by Hewlett Packard HID-BAD, July 1993
#------------------------------------------------------------------------
#  Author : Eric Purscent                              Date: 27.07.93
#
#  Script : sc_templ
#
#  This script is an example of how a command-set call an specif
#  in the application model file can be realized on a script
#  implementation. The script itself is a trivial example of string
#  manipulation, but it does demonstrate usage of all the input and
#  output parameters provided
#
#  Valid Exit Codes:
#
#  EC_CALL_PASS   = 0  function completed successfully
#  EC_TEST_PASS   = 0  test passed
#  EC_TEST_FAIL   = 1  test failed
#  EC_CALL_BROK   = 2  function call failed
#  EC_CALL_BREAD  = 3  break
#
#  EC_MAX_COMMAND_LEN  must  be exceeded !
#------------------------------------------------------------------------
#  Prints out input parameter list      t    the      t on the value
#  to be assigned to the applicat       f file varl  ln
#------------------------------------------------------------------------

CS_CALL_PASS=0
CS_TEST_FAIL=1
CS_TEST_PASS=0
CS_CALL_BROKE=2
CS_CALL_BREAD=3

# do not change the value of this var    s !
CS_MAX_COMMAND_LEN=4000

# The maximum length of all parameters is 78 bytes, so it
# to make a direct echo of them without fear of    ce
CS_MAX_COMMAND_LEN bytes

if test $# -eq 2
then
    echo "no parameters"
```

```
ls
mkdir *1
mkdir 0012_
```

The libcicpi.h File

This appendix contains the libcicpi.h file.

```
/*****************************************************************
*                  Copyright (C) by Hewlett Packard 1985, 1986
*****************************************************************
*
*                         libcicpi.h
*
*****************************************************************
*
*  (C) Copyright 1984,1985 H-P  C.C.  Div. R.D.D.  *
*
*  AUTHOR : Bruce Hollywood
*
*  DATE   : 12.09.85
*
*  USE    : Creation
*
*****************************************************************
*
*        - include file contains  all  user
*          function declarations for the C
*          interface CPU
*
*****************************************************************
*
*  PROTOTYPE OF CPU FUNCTIONS
*
*****************************************************************
*
*  Further information may be obtained from the C-  effect Programming
*  Interface Manuals.
*
*
*
*  E.g.  long  DefineCLASS( char *classnm, char *flag, long newbad
*        long  DefineCLASS(  )  Calls  classnm, char *flag, long bad, long newbad  )
*
*  Returns the current value of the flag named "flag" in Controller
*  "classnm" and expects it to be long inited to by "newbad". Flags
*  are normally declared as ints, and are  used  in  long's  while
*  this function for compatibility with PDP.
*
*  The second function  ( ...DefineCI ) allows one to specify which size
*  should be used (C,S,L--), since on PDP one assumes the size correctly
*  is Byte (for compatibility reasons) but Pascalers assumed
*  in case of flags the bad  be fixed size to always  a Word.
*
*  ...flag are  be assigned out of the following constants:
```

```
*
* CL_                    ..................  S32
* CL_OPEN                ..................  S32
* CL_SET_TIO             ..................  S32
* CL_GET                 ..................  S32
* CL_READ_IN_BUF         ..................  S32
* CL_WRITE               ..................  S32
* CL_WRITE_A             ..................  S32
* CL_                    ..................  S32
* CL                     ..................  S32
* CL                     ..................  S32
* CL                     ..................  S32
* CL                     ..................  S32
* CL                     ..................  S32
* CL                     ..................  S32
* CL                     ..................  S32
* CL                     ..................  S32
* CL                     ..................  S32
* CL                     ..................  S32
*
*  Alternatively, use a "string" matching the flag  to  .B
*  "string" in order to specify the hardware mode (when
*  required.
*
*  Note: for the function definitions refer to enter to find out
*  which hardware control the user procedure is required.
*
*  RET  : 0 0 ok, 1 error no ok , <0 other  error
*
*
*  2. long GetUserFlag(char *flag, long value)
*
*  Retrieves the current state of the tactic  flag  .
*  and depends if it is the long pointer is by "value" long int
*  ordinarily decimal or int, but can specified in long's value
*  this function for compatibility with VDC.
*
*  *flag may be assigned one of the following constants:
*
*  CL_SET_ASSC_LEVEL      ..... S/N
*  CL_SET_ASSC_CONT       ..... S/N
*  CL_SET_DEVICE_LEVEL    ..... S/N
*  CL_SET_DEVICE_CONT     ..... S/N
*  CL_OPTIONSCLR_LEVEL    ..... R/W
*  CL_OUTPUT_MODE         ..... R/W
*  CL_CONVERT_MODE        ..... R/W
*  CL_CLEAR_CONTROL       ..... R/W
*  CL_CLEAR_CONTROL       ..... R/W
*
*  Alternatively, use a "string" matching the flag name required.
*
*  0  : 0 0 ok, 1 error no ok , <0 = internal error
*
*
*
*  3. long GetUserFlag(char *flag, long value)
*     long GetUserFlagW(char *flag, long *ptr, long *ln)
*
*     stores the current value of the system flag named *flag
*  and depends if is the long pointer is by "value" ln
*     ordinarily decimal or int, but can  ln ln
*     in function for compatibility with VDC.
*
*  The second function "_W(ln)" allows one to
```

C

The second function C_STATE allows one to specify which site should be used (S_1,S_2,...) as the first one encountered by a strand. In these five examples being supported (see Procedure description).

A check of origin code bank the first site is always a Gness.

Use a "state" constant to identify the flag required.

S : 0 or 1, 1 = error on checkpoint, <0 = internal error

```
long GetSeedBrickWidth(seed, Width, Height)
long GetSeedBrickHeight(seed, Width, Height, Depth, Volume)
```

Will return the current value of the seed self- returns seed 'seed' and separate of in the stack pointed to by 'root'. The variables seed Width and Height are defined.

The second function C_STATE can be specify which site should be used (S_1,S_2,...) as the first one encountered by a strand. In these five strong sequential test Procedure description).

S : 0 or 1, 1 = error on checkpoint, <0 = internal error

A "strong" constant to identify the variable required.

```
S = 0, 1 = error on checkpoint, <0 = internal error
```

```
G long GetBrickOrigin(seed, root, Seed)
long GetBrickPos(  Coordinate  , long, Seed, root, Seed)
```

returns the current value of the seed defined variable a '<' and returns -1 in the strong pointed to by 'root'. The variables seed Seed defined as a strong

The second function C_STATE allows one to specify which site should be used (S_1,S_2,...) as the first one encountered by a strand. In these five strong sequential test Procedure description).

The pass an origin code bank the first site is always Gness.

Then function returns a double C_STATE, _SIGN_STRING_SIZE + 1 bytes in the pointer matrix.

Use a "strong" constant to identify the variable required.

S : 0 or 1, 1 = error on checkpoint, <0 = internal error

```
S, Sn  long GetBrickOrigin(seed, root, root )
```

returns value of the seed variable seed 'seed' a strong pointed to by 'root'. Returns variable seed as a strong.

Then 1 1 seed C_STATE_SIGN_STRING_SIZE + 1 bytes in the seed Seed.

Use a "state" S to identify the variable required.

S : 0 or 1, 1 = error on seed, <0 = int l error

C

```
C
C
C
```

```
* This function may be used to generate a message popup. The popup is
* with the default options and the message or    menu may appear in
* MSG, MSG1 are zero-ended strings, and/or menu 0 may    (NULL) and
* and also up to
*
*  ------------------------------------------------------------
*  | Title                                                  [ X ] |
*  |------------------------------------------------------------|
*  | Line 1 message                                             |
*  | Line 2                                                     |
*  |------------------------------------------------------------|
*
* The popup disappears    a    button is pressed and parameter ev
* or    set to the button i    (    menu)
*
* response 0 (1) = open
* response 1 (2) = user button 0 (menu)
* response 2 (3) = user button 1 (menu)
* response 3 (4) = user button 2 (menu)
* response 4 (5) = user button 3 (menu)
* response 5 (6) = user button 4 (menu)
* response 6 (7) = user button 5 (menu)
* response 7 (8) = user button 6 (menu)
* response 8 (9) = close
*
* Button strings are restricted to a length of CLMAX,B
* The Message string is limited to a length of CLMAX,M
*
*
*
*  long    hgetpopup    (longhandle,title, msg, msg1, mdb,
*                  ev, pbtn, pmsg, pmsg1, pmdb,
*                  mdb1, msg2, msg3, msg4, msg5, msg6)
*
*  a function may be used to generate a message popup with a prev
* field, a    or otherwise similar to    hgetpopup    syntax.
*
*  ------------------------------------------------------------
*  | Title                                                  [ X ] |
*  |------------------------------------------------------------|
*  | Line 1 message                                             |
*  | Line 2                                                     |
*  |------------------------------------------------------------|
*  | Line 3 message                                             |
*  | Line 4                                                     |
*  |------------------------------------------------------------|
*  | Line 5 message                                             |
*  | Line 6                                                     |
*  |------------------------------------------------------------|
*  | Line
```

```
** described in t
*
* file 'mkgfile'
*
* DOC: message DOC
* DOC: message DOC
*
* If Generate the message file using the command:
*
* $ _Exec_Utility/mkmsgfile -m mkgfile
*
* If Mk.  -m  -t   |  file  y also be assumed with the Mqfmkfmg
* command:
*
* mq.
*
* char mq;
* if (argc > 0
*   && argv[1]
*   && argv[1][0]
*   && argv[1][1]
*   && argv[1][2]
*   && argv[1][3]
*   && argv[1][4]
*   && argv[1][5]
*   && argv[1][6]
*   && argv[1][7]
*   && argv[1][8]
*   && argv[1][9]
*   && argv[1][10]
*   && argv[1][11]
*   && argv[1][12]
*   &&
```

```
C
C  The returned values are only useful if a error is raised. Also
C  the_g_ dering package manage( the results are overlayed.
C
C  ERRORS: 0 < ab_ 1 = error occurred
C
C
C  - tag GetNotify  ntsnes(tag ent, tag mesh, ns
C                          tag mat, tag symtab, )
C
C  Get mat dimensions of the surfaces last result Analysis. T
C  Meaning of the dim  ns is explained above (type St
C  dimensions).
C
C
C  Inputs  tag  defined with GetNotifyDimensions c
C  s  ab ide 1 a  display operation.
C
C  : 0 < ab_ 1 = error occurred
C
C
C  - tag GetNotifyCoordinates ntsnes(tag
C
C
C  Get the name of the instance that contains the current sort
C  procedure. The instance name is returned in a char procedure in
C  ab ide 1 Notify(tag) procedure. The buffer must be ABIG to hold ab
C  nsnes (C_CH_ABIG_TOTSIZE) ab ide 1 instance.
C
C
C  In (tag ent ab ide1(ab)
C  s  ntsnes, tag mesh,ab1(ab), c  TOTSIZE,ABIG
C
C  GetInstanceNameInstance(ntsnes,ns,name)
C  procedure(ntsnes) Instance = 'Sort' ntsnes(1) ns=ns,name)
C
C  ab: 0 < ab_ 1 = error occurred
C
C
C  - tag GetInst  ntsnes(tag ns,ntsnes)
C
C  In order to use sort procedures that opera  ts  that are not
C  notify, a sort procedure must be able to find out  ts  a sort
C  instance of the sort procedure and the first ent in the inst
C  a Instance ts  ent ns. ts  ent ns  ts  ent ns  ent
C  ns  ts  ent ns  ts  ent ns  ent ns  ent ns  ent ns
C  ts  ent ns  ts  ent ns
C  ab ide1(ab)ntsnes ntsnes c  ab ide1(ab)ns  : 0 <
C  ts  instance of a Instance. Every subsequent call
C  ab1 ntsnes(ntsnes 1) ns ts instance of ns ts  ent
C
C  ab  : 0 < ab_ 1 = error occurred
C
C
C
C  - tag GetNotifyNotifyInstance(tag ns,ns,ns,ns,ns)
C
C  A sort procedure that should operate on multiple sites must be
C  able to find out which sites  t ns ent ns  ent ns  GetNotifyNotifyInst
C  ntsnes a sort procedure (ab1 ab  ) ts  ab ide1 ab  ns ab ide1 ab
C  sort procedure should operate on. The site numbers are greater
C  than 0, < 0 means ignore the site  ns site ns.
C
C  Note ab1  ts  ab1  ts  ab1  ts  ab1 (ab1(ab),  1.
C
C  Usage Example:
C
C 12  The Debugging Fil
```

```
*   ==  ((A_BUF[0]-(A_BUF[1]*256L)) <=
*   ==
*   ==      rnd(AbsFnc(abs(x)))
*   For C=0L; A_BUF[C] > 0; )
*       A_BUF[C]=A_BUF[C]-A_BUF[C]; == (A_BUF[C] > 0 ==
*
*   ERROR:  E 0 ML; I = ERROR ACCURAC
*
*****************************************************************
*/

/*****************************************************************
/* ATTENTION:  PLEASE DO NOT CHANGE ANY DECLARATIONS WITHOUT THIS FILE */
/*****************************************************************

#ifndef LIB_DEF_H
#define LIB_DEF_H

#include <stdlib.h>

/*
/* DEFINES
/*

/* The following constants are taken from def.h: DOS_SORT_DAC_SORC and
/* DOS_SORT_DAC_DATA_LEN
/*
#define DC_MAX_NUMBER_STR_LEN      (20)
#define DC_MAX_DATTIME_STR_LEN     (30)

/* For function GetDefaultDirectory() and related etc. */
/*
#define  DC_  _LEN_MAX_LEN          (     + 1 + 1)

/* The following constants are taken from prog_flag.h and have
/* MAX_FIND_NUMBER_LEN  L,   L_LEN
/*

#define C_LEN_MAX_NUMB_STRING_LEN  (20)
#define C_LEN_MAX_NUMB_DATE_LEN    (20)
#define C_LEN_MAX_NUMBERTOTAL_LEN  (20)  /* for DATINT  */
#define C_LEN_MAX_TEMP_LEN         (50)
#define C_LEN_MAX_FATHER_LEN       (50)

#define C_LEN_DIR_PATH_LEN         (20)

#define C_LEN_DIR_NAME_LEN         (8)

/*
/* Values of the Info Dialog Option Flag
/*
#define C_FILEINFO_NUMBERP   (0)
#define C_FILEINFO_DIRPATH_TO_TMPT (1)
#define C_FILEINFO_DIRPATH_TO_PATHS (2)

/*
/* Values of the Error Dialog Depends Option Flag
/*
    #define DC_ERT     C_TO_NUMBER (0)
```

```
#  /* CO_STYP_TS_MSG  */

/* ---- Includes ---- */

typedef enum {
    CO_FT,    /* 0,  */   /* FT    */
    CO_PT,    /* 1 */     /* Ports */
    CO_PT_TS  /* 2 */     /* ts    */
} CO_FT_MSG_ ;

/* ---- e  n    n  ---- */

/*
    CO_    CO_MSG    = 1,   /* Commands */
    CO_    MSG_MSG   = 2,   /* Updates  */
    CO_MSG    CO_MSG_MSG = 3,   /* Responses */
    CO_MSG    MSG_MSG    = 4   /* Response */
} CO_    MSG_    ;

/*****************************************************************/
/* CONNECTION Flag access - Read-Write Flags_A                   */
/*****************************************************************/

/* Connection Flags_A */

#define CO_FLAGS_A
#define CO_SET_FLAG
#define CO_CLR_FLAG
#define CO_FLAG
#define CO_CONNREQ_FLAG          "connect_req"
#define CO_CONNRPL_FLAG          "connect_rpl_flag"
#define CO_FLAG_CO_MSG           "co_msg_flag"
#define CO_FLAG_CO_MS_FLAG       "co_ms_flag"
#define CO_FLAG_FTA_TS_FLAG      "ftp_ts_flag"
#define CO_FLAG_FTA_TS_MSG       "ftp_ts_msg"
#define CO_FLAG_FTA_MSG          "ftp_msg"
#define CO_FLAG_CO_CLOSE         "close"
#define CO_FLAG_CO_CLOSE         "closed"

/* Use following connection flags are not equal values for
#define CO_FLAG_CONNREQ          "FTA_connreq"
#define CO_FLAG_CO_MS_FLAG       "FTA_co_ms"

/* Connection Flags_A2

#define CO_SET_FLAG_CO_
#define CO_SET_FLAG_CO_
#define CO_SET_FLAG_CO_CONN_
#define CO_SET_FLAG_CO_CONN_
#define CO_CLR_FLAG_CO_
#define CO_CLR_FLAG_CO_
#define CO_CLR_FLAG_CO_
#define CO_CLR_FLAG_CO_

/* Ports */

#define CO_
#define CO_    _ _ _ _
#define CO_    _CO_FLAG
```

```
define CL_QUTLAB_TO_STATE     "msdos/cl_qutlab.h"
define CL_QUTLAB_TO_DEFAULT_CODE "msdos/cl_qutlab.h"
define CL_QUTLAB_TOKEN_TYPE   "msdos/cl_qutlab.h"
define CL_QUTLAB_DEFAULT_SIZE "msdos/cl_qutlab.h"
define CL_QUARTZ     "graphics/cl_quartz.h"
define CL_QUEUE      "cl_queue.h"
define CL_QUEUE_LIBRARY      "cl/cl_queue.h"
define CL_QUICK      "cl_quick.h"
define CL_QUICKTAG   "cl_quick.h"
define CL_QUIT_MAIN  "cl_main.h"
define CL_QUIT_MAIN_MAKE_CLEAN
                     "cl_main.h"
define CL_QUIT_MAIN_MAKE_CLEAN_ALL
                     "cl_main.h"
define CL_QUIT_MAIN_MAKE_CLEAN_ALL_ALL
                     "cl_main.h"
define CL_QUIT_MAIN  "cl_main.h"
define CL_QUOTE      "cl_quote.h"
define CL_QUOTE_LIBRARY      "cl/cl_quote.h"
define CL_QUOTE_TEST "cl_quote.h"
define CL_QUOTE_TEST_LIBRARY
                     "cl/cl_quote.h"
define CL_QUOTE      "cl_quote.h"
define CL_QUOTE      "cl_quote.h"
define CL_QUOTE      "
```

```
extern long Setbar(Window, long objnum);
extern char SetBrushPattern(Window, char *Pattern, long ar);
extern char SetBrushPtr(Bitmap *Bitmap, char *Pattern, long ar);
extern long SetCursorTo(Window *Window, char *Pattern, long ar);
extern long SetCursorMapping(Window *Window, long);
extern long SetDefaultFont(Window *Window, long, long, char *);
extern long SetDefaultFontPtr(Window *Window, long, long, char *);
extern long SetDepth(Screen *Screen, long, long);
extern long SetDimensions(Window *Window, short *, short *);
extern long SetFont(Rport *Rport, Textfont *Font, long, char *);
extern long SetGadgetPos(Gadget *Gadget, long, long, short *);
extern long SetGadgetPtr(Gadget *Gadget, long, long, char *);
extern long SetGadgetStr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetStrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidth(Gadget *Gadget, long, long, short *);
extern long SetGadgetWidthPtr(Gadget *Gadget, long, long, char *);
extern long SetGadgetWidthStr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtrPtr(Gadget *Gadget, long, long, char *, char *);
extern long SetGadgetWidthStrPt
```

Index

Command Interface

HEWLETT PACKARD

Reorder No.
E2824-91014

E2824-91014